2

Programmation Shell

Avant d'entamer la programmation sous shell, rappelons la syntaxe élémentaire qui est utilisée pour lancer des processus depuis la ligne de commande sous Unix et Linux, puisque nous emploierons également ces symboles dans les scripts.

Saisie	Signification
`commande`	La commande est exécutée normalement ; le symbole d'invite sera affiché lorsqu'elle se terminera.

`commande &`	La commande est exécutée en arrière-plan, ce qui signifie qu'elle n'a en principe pas accès au terminal. Le shell reprend donc la main immédiatement, et affiche son symbole d'invite alors que la commande continue à s'exécuter en tâche de fond. Le shell affiche une ligne du type
	`[1] 2496`
	indiquant le numéro du job à l'arrière-plan, suivi du PID (l'identifiant de processus). Lorsque la commande se termine, le shell affichera une ligne
	`[1]+  Done commande`
	juste avant de proposer un nouveau symbole d'invite.

`commande > fichier`	La commande est exécutée, mais sa sortie standard est dirigée vers le fichier indiqué. Seule la sortie d'erreur s'affiche à l'écran.
`commande >> fichier`	La sortie standard est ajoutée en fin de fichier sans en écraser le contenu.
`commande < fichier`	La commande est exécutée, mais ses informations d'entrée seront lues depuis le fichier qui est indiqué plutôt que depuis le terminal.
`cmmnd_1 | cmmnd_2`	Les deux commandes sont exécutées simultanément, l'entrée standard de la seconde étant connectée par un tube (*pipe*) à la sortie standard de la première.

Nous étendrons ces possibilités ultérieurement, mais ces éléments seront déjà suffisants pour comprendre les premiers scripts étudiés.

Premier aperçu

Avant d'entrer dans le détail de la syntaxe et des commandes pour le shell, nous allons tout d'abord survoler un script complet, afin de nous familiariser avec la structure des programmes shell. Ce script, déjà assez important pour un premier exemple, est une variation sur un article que j'avais écrit en mai 1996 pour le journal à la ligne *Linux Gazette, [Blaess 1996]*.

Premier script, rm_secure

L'idée est de remplacer la commande `rm` normale (suppression de fichiers et de répertoires) par un script qui permette de récupérer les fichiers effacés malencontreusement. Il en existe des implémentations plus performantes, mais l'avantage de ce script est d'être assez simple, facile à installer et à étudier. Nous allons l'analyser en détail, dans son intégralité. Des numéros ont été ajoutés en début de ligne pour référence, mais ils ne font évidemment pas partie du script.

rm_secure.sh :

```
 1  sauvegarde_rm=~/.rm_saved/
 2
 3  function rm
 4  {
 5    local opt_force=0
 6    local opt_interactive=0
 7    local opt_recursive=0
 8    local opt_verbose=0
 9    local opt_empty=0
10    local opt_list=0
11    local opt_restore=0
12    local opt
13
14    OPTERR=0
15    # Analyse des arguments de la ligne de commande
16    while getopts ":dfirRvels-:" opt ; do
17      case $opt in
18        d ) ;; # ignorée
19        f ) opt_force=1 ;;
20        i ) opt_interactive=1 ;;
21        r | R ) opt_recursive=1 ;;
22        e ) opt_empty=1 ;;
23        l ) opt_list=1 ;;
24        s ) opt_restore=1 ;;
25        v ) opt_verbose=1 ;;
26        - ) case $OPTARG in
27              directory )     ;;
28              force)        opt_force=1 ;;
29              interactive ) opt_interactive=1 ;;
30              recursive ) opt_recursive=1 ;;
31              verbose )   opt_verbose=1 ;;
```

```
32              help )      /bin/rm --help
33                echo "rm_secure:"
34                echo "  -e  --empty    vider la corbeille"
35                echo "  -l  --list     voir les fichiers sauvés"
36                echo "  -s, --restore  récupérer des fichiers"
37                return 0 ;;
38              version ) /bin/rm --version
39                echo "(rm_secure 1.2)"
40                return 0 ;;
41              empty )   opt_empty=1 ;;
42              list )    opt_list=1 ;;
43              restore ) opt_restore=1 ;;
44              * )  echo "option illégale --$OPTARG"
45                   return 1;;
46            esac ;;
47        ? ) echo "option illégale -$OPTARG"
48            return 1;;
49      esac
50    done
51    shift $(($OPTIND - 1))
52
53    # Créer éventuellement le répertoire
54    if [ ! -d "$sauvegarde_rm" ] ; then
55      mkdir "$sauvegarde_rm"
56    fi
57
58    # Vider la poubelle
59    if [ $opt_empty -ne 0 ] ; then
60      /bin/rm -rf "$sauvegarde_rm"
61      return 0
62    fi
63
64    # Liste des fichiers sauvés
65    if [ $opt_list -ne 0 ] ; then
66        ( cd "$sauvegarde_rm"
67          ls -lRa * )
68    fi
69
70    # Récupération de fichiers
71    if [ $opt_restore -ne 0 ] ; then
72      while [ -n "$1" ] ; do
73        mv "${sauvegarde_rm}/$1" .
74        shift
75      done
76      return
77    fi
78
79    # Suppression de fichiers
80    while [ -n "$1" ] ; do
81    # Suppression interactive : interroger l'utilisateur
82      if [ $opt_force -ne 1 ] && [ $opt_interactive -ne 0 ]
        then
```

```
 83       local reponse
 84       echo -n "Détruire $1 ? "
 85       read reponse
 86       if [ "$reponse" != "y" ] && [ "$reponse" != "Y" ] &&
 87          [ "$reponse" != "o" ] && [ "$reponse" != "O" ] ; then
 88         shift
 89         continue
 90       fi
 91     fi
 92     if [ -d "$1" ] && [ $opt_recursive -eq 0 ] ; then
 93       # Les répertoires nécessitent l'option récursive
 94       shift
 95       continue
 96     fi
 97     if [ $opt_verbose -ne 0 ] ; then
 98       echo "Suppression $1"
 99     fi
100     mv -f "$1" "${sauvegarde_rm}/"
101     shift
102   done
103   }
104
105   trap "/bin/rm -rf $sauvegarde_rm" EXIT
```

L'analyse détaillée du script est un peu laborieuse, mais il est nettement préférable d'étudier un programme réel, complet et utilisable, qu'un exemple simpliste, construit uniquement pour illustrer les possibilités du shell. Si certaines parties semblent obscures, il n'y a pas lieu de s'inquiéter : nous y reviendrons dans la présentation des commandes et de la syntaxe.

Analyse détaillée

Exceptionnellement, ce script ne débute pas par une ligne shebang introduite par `#!`. En effet, il n'est pas conçu pour être exécuté dans un processus indépendant avec une commande comme `./rm_secure.sh`, mais pour être lu par une commande `source` ou `.` depuis un fichier d'initialisation du shell (comme `~/.profile` ou `~/.bashrc`).

Lorsqu'un script est appelé ainsi « `. script` », il est interprété directement par le shell en cours d'exécution, sans lancer de nouvelle instance. Aussi les variables et fonctions définies par ce script seront-elles disponibles même après sa terminaison. Il ne faut pas confondre le point utilisé ici pour « *sourcer* » un script, et le point représentant le répertoire courant dans l'appel `./script`.

Ce script définit une fonction nommée `rm`, s'étendant des lignes 3 à 103. Le programme étant lu et interprété directement par le shell, et non par un processus fils, cette fonction sera disponible dans l'environnement du shell qui vient de s'initialiser.

Lorsqu'on appelle, depuis la ligne de saisie, un nom de commande comme `rm`, le shell va d'abord contrôler s'il existe dans son environnement une fonction qui a ce nom. S'il n'en

trouve pas, il vérifiera s'il s'agit de l'une de ses commandes internes (comme `cd`). En dernier ressort, il parcourra l'ensemble des répertoires mentionnés dans la variable d'environnement `PATH` à la recherche d'un fichier exécutable qui ait le nom indiqué. Naturellement, si la commande contient un chemin d'accès (par exemple `/bin/rm`), il évite tout ce mécanisme et lance directement le fichier désiré.

Dans notre cas, nous définissons une fonction ayant pour nom `rm`, qui sera donc invoquée à la place du fichier `/bin/rm` habituel.

La première ligne du script définit une variable qui contient le nom d'un répertoire vers lequel les fichiers seront déplacés plutôt que véritablement effacés. Des options à la ligne de commande nous permettront d'examiner le contenu du répertoire et de récupérer des fichiers. Il est évident que l'utilisateur doit avoir un droit d'écriture sur ce répertoire. La configuration la plus intuitive consiste à utiliser un répertoire qui figure déjà dans le répertoire personnel de l'utilisateur et de lui donner un nom qui commence par un point afin d'éviter de le retrouver à chaque invocation de `ls`.

Nous voyons que l'affectation d'une variable se fait, avec le shell, simplement à l'aide d'une commande `nom=valeur`. Il est important de noter qu'il ne doit pas y avoir d'espace autour du signe égal.

Vient ensuite la déclaration de la fonction. On emploie le mot-clé `function` suivi du nom de la routine, puis de son corps encadré par des accolades. Les lignes 5 à 12 déclarent des variables locales de la fonction. Ces variables serviront à noter, lors de l'analyse de la ligne de commande, les différentes options indiquées par l'utilisateur. Les déclarations sont ici précédées du mot-clé `local`, qui permet de ne les rendre visibles que dans notre fonction. Quand une fonction est, comme ici, destinée à rester durablement dans la mémoire du shell, il est important de ne pas « polluer » l'environnement en laissant inutilement des variables apparaître en dehors de la routine où elles sont employées.

Notre fonction `rm` accepte les mêmes options à la ligne de commande que la commande `/bin/rm`, et en ajoute trois nouvelles :

Option	Signification	Provenance
`-d ou --directory`	Ignorée	`/bin/rm`
`-f ou --force`	Ne pas interroger l'utilisateur	`/bin/rm`
`-i ou --interactive`	Interroger l'utilisateur avant de supprimer un fichier	`/bin/rm`
`-r, -R, ou --recursive`	Supprimer d'une façon récursive les répertoires	`/bin/rm`
`-v ou --verbose`	Afficher les noms des fichiers avant de les supprimer	`/bin/rm`
`--help`	Afficher une page d'aide	`/bin/rm`
`--version`	Afficher le numéro de version du programme	`/bin/rm`
`-e ou --empty`	Vider la corbeille de sauvegarde	`rm_secure`
`-l ou --list`	Voir le contenu de la corbeille	`rm_secure`
`-s ou --restore`	Récupérer un fichier dans la corbeille	`rm_secure`

Avant de lire les arguments de la ligne de commande, nous devons initialiser une variable système nommée `OPTIND`, ce qui a lieu à la ligne 14. Le rôle de cette variable sera détaillé plus bas.

La ligne 16 introduit une boucle `while`. Voici la syntaxe de ce type de boucle :

```
while condition
do
    corps de la boucle...
done
```

Les instructions contenues dans le corps de la boucle sont répétées tant que la condition indiquée est vraie. Afin de rendre l'écriture un peu plus compacte, il est fréquent de remplacer le saut de ligne qui se trouve après la condition par un caractère point-virgule, qui, en programmation shell, signifie « fin d'instruction ». La boucle est donc généralement écrite ainsi :

```
while condition ; do
    corps de la boucle
done
```

Une erreur fréquente chez les débutants est de tenter de placer le point-virgule là où il paraît visuellement le plus naturel pour un programmeur C ou Pascal : en fin de ligne. Sa position correcte est pourtant bien avant le `do` !

Ici, la condition est exprimée par une commande interne du shell :

```
getopts ":dfirRvels-:" opt
```

La commande `getopts` permet d'analyser les arguments qui se trouvent sur la ligne de commande. Nous lui fournissons une chaîne de caractères qui contient les lettres attribuées aux options (comme dans le tableau de la page précédente), et le nom d'une variable qu'elle devra remplir. Cette fonction renvoie une valeur vraie tant qu'elle trouve une option qui est contenue dans la liste, et place le caractère dans la variable `opt`. Nous détaillerons ultérieurement le fonctionnement de cette routine, et la signification des deux-points présents dans cette chaîne. Retenons simplement qu'arrivés à la ligne 17, nous savons que l'un des caractères inscrits dans la chaîne est présent dans la variable `opt`.

Sur cette ligne 17, nous trouvons une structure de sélection `case` qui va nous permettre de distribuer le contrôle du programme en fonction du contenu d'une variable. Sa syntaxe est la suivante :

```
case expression in
    valeur_1 )
        action_1 ;;
    valeur_2 )
        action_2 ;;
esac
```

La valeur renvoyée par l'expression, ici le contenu de la variable `opt`, est comparée successivement aux valeurs proposées jusqu'à ce qu'il en soit trouvé une qui lui corresponde, et l'action associée est alors exécutée. Les différents cas sont séparés par deux

points-virgules. Pour le dernier cas, toutefois, ce signe de ponctuation est facultatif, mais il est recommandé de l'employer quand même, afin d'éviter des problèmes ultérieurs si un nouveau cas doit être ajouté. Le mot-clé `esac` (`case` à l'envers), que l'on trouve à la ligne 49, sert à marquer la fin de cette structure de sélection.

Nous remarquons au passage que, ligne 17, nous lisons pour la première fois le contenu d'une variable. Cela s'effectue en préfixant son nom par un caractère `$`. Ainsi, nous nous référons au contenu de la variable `opt` en écrivant `$opt`. Il faut bien comprendre dès à présent que le nom réel de la variable est bien `opt` ; le `$` est un opérateur demandant d'accéder à son contenu. Nous verrons plus avant des formes plus générales de cet opérateur.

Les valeurs auxquelles l'expression est comparée dans les différentes branches de la structure `case` ne sont pas limitées à de simples valeurs entières, comme peuvent y être habitués les utilisateurs du langage C, mais peuvent représenter des chaînes de caractères complètes, incorporant des caractères génériques comme l'astérisque `*`. Ici, nous nous limiterons à employer le caractère | qui représente un *OU* logique, à la ligne 21. Ainsi l'action

```
opt_recursive=1
```

est exécutée si le caractère contenu dans `opt` est un `r` ou un `R`.

Nous observons que les lignes 19 à 25 servent à remplir les variables qui représentent les options en fonction des caractères rencontrés par `getopts`.

Un cas intéressant se présente à la ligne 26, puisque nous avons rencontré un caractère d'option constitué par un tiret. Cela signifie que l'option commence par « `--` ». Dans la chaîne de caractères que nous avons transmise à `getopts` à la ligne 16, nous avons fait suivre le tiret d'un deux-points. Cela indique à la commande `getopts` que nous attendons un argument à l'option « `-` ». Il s'agit d'une astuce pour traiter les options longues comme `--help`. Lorsque `getopts` rencontre l'option « `-` », il place l'argument qui la suit (par exemple `help`) dans la variable `OPTARG`. Nous pouvons donc reprendre une structure de sélection `case` pour examiner le contenu de cette variable, ce qui est fait de la ligne 26 à la ligne 46 où se trouve le `esac` correspondant.

Les options longues, `directory`, `force`, `interactive`, `verbose`, `recursive`, `empty`, `list`, `restore`, sont traitées comme leurs homologues courtes, décrites dans le tableau précédent. Aussi les options `help` et `version` commencent par invoquer la véritable commande `rm` afin qu'elle affiche ses propres informations, avant d'écrire leurs messages à l'aide de la commande `echo`. On remarquera que ces deux cas se terminent par une commande `return 0`, ce qui provoque la fin de la fonction `rm`, et renvoie un code de retour nul indiquant que tout s'est bien passé.

Le motif mis en comparaison sur la ligne 44, un simple astérisque, peut correspondre à n'importe quelle chaîne de caractères selon les critères du shell. Ce motif sert donc à absorber toutes les saisies non valides dans les options longues. Nous affichons dans ce cas un message d'erreur au moyen de la commande :

```
echo "option illégale --$OPTARG"
```

Nous remarquons qu'au sein de la chaîne de caractères encadrée par des guillemets, l'opérateur `$` agit toujours, et remplace `OPTARG` par son contenu, c'est-à-dire le texte de l'option longue erronée. En ce cas, nous invoquons la commande `return` avec un argument `1`, ce qui signifie conventionnellement qu'une erreur s'est produite.

À la ligne 47, nous revenons à la première structure de distribution `case-esac`. Nous avions mis à la ligne 16 un deux-points en première position de la chaîne d'options transmise à `getopts`. Cela permet de configurer le comportement de cette commande lorsqu'elle rencontre une lettre d'option non valide. Dans ce cas, elle met le caractère `?` dans la variable indiquée, `opt` en l'occurrence, stocke la lettre non valide dans la variable `OPTARG`, et n'affiche aucun message d'erreur. Notre dernier cas, sur la ligne 47, sert donc à afficher un message d'erreur, et à arrêter la fonction avec un code d'échec.

Après clôture avec `esac` et `done` de la lecture des options, il nous reste à traiter les autres arguments qui se trouvent sur la ligne de commande, ceux qui ne sont pas des options, c'est-à-dire les noms de fichiers ou de répertoires qu'il faut supprimer ou récupérer.

Lorsqu'un script (ou une fonction) est invoqué, les arguments lui sont transmis dans des variables spéciales. Ces variables, accessibles uniquement en lecture, sont représentées par le numéro de l'argument sur la ligne de commande. Ainsi, `$1` renvoie la valeur du premier argument, `$2` celle du deuxième, et ainsi de suite. Nous verrons ultérieurement qu'il existe aussi des variables spéciales, accessibles avec `$0`, `$#`, `$*` et `$@`, qui aident à manipuler ces arguments.

La boucle `while` autour de `getopts` a parcouru toutes les options qui se trouvent en début de ligne de commande, puis elle se termine dès que `getopts` rencontre un argument qui ne commence pas par un tiret. Avant d'analyser un argument, `getopts` stocke son indice dans la variable `OPTIND`, que nous avions initialisée à zéro à la ligne 14. Si les *n* premiers arguments de la ligne de commande sont des options valides, en sortie de la boucle `while`, la variable `OPTIND` vaut *n+1*. Il est donc à présent nécessaire de sauter les `OPTIND-`*1* premiers arguments pour passer aux noms de fichiers ou de répertoires. C'est ce qui est réalisé sur la ligne 51 :

```
shift $(($OPTIND - 1))
```

La construction `$(( ))` encadre une expression arithmétique que le shell doit interpréter. Ici, il s'agit donc simplement du nombre d'arguments à éliminer, soit `$OPTIND-1`. La commande `shift` *n* décale les arguments en supprimant les *n* premiers.

Nous allons entrer à présent dans le vif du sujet, avec les fonctionnalités véritables du script. Toutefois, nous allons auparavant créer le répertoire de sauvegarde, s'il n'existe pas encore. Cela nous évitera de devoir vérifier sa présence à plusieurs reprises dans le reste du programme. Nous voyons donc apparaître sur les lignes 54 à 56 une nouvelle structure de contrôle, le test `if-then-else`. Sa syntaxe est la suivante :

```
if condition
then
    action
else
    autre action
fi
```

On regroupe souvent les deux premières lignes en une seule, en séparant les deux commandes par un point-virgule, comme ceci :

```
if condition ; then
    action
else
    autre action
fi
```

Si la condition est évaluée à une valeur vraie, la première action est exécutée. Sinon, le contrôle passe à la seconde partie, après le `else`. Dans notre cas, la condition est représentée par une expression a priori surprenante :

```
[ ! -d "$sauvegarde_rm" ]
```

En fait, la construction `[ ]` représente un test. Il existe d'ailleurs un synonyme de cette construction qui s'écrit `test`. On pourrait donc formuler notre expression comme ceci :

```
test ! -d "$sauvegarde_rm"
```

L'emploi des crochets est plus répandu, peut-être par souci de concision. On notera qu'il ne s'agit pas simplement de caractères d'encadrement comme `( )` ou `{ }`, mais bien d'une construction logique complète, chaque caractère étant indépendant. Cela signifie que les deux crochets doivent être entourés d'espaces. Il ne faut pas essayer de les coller à leur contenu comme on pourrait avoir tendance à le faire : `[! -d "$sauvegarde_rm"]`, le shell ne le permet pas.

L'option `–d` du test permet de vérifier si le nom fourni en argument est bien un répertoire existant. Le point d'exclamation qui se trouve au début sert à inverser le résultat du test. Les lignes 54 à 56 peuvent donc s'exprimer ainsi :

si le répertoire `$sauvegarde_rm` *n'existe pas, alors*

```
mkdir "$sauvegarde_rm"
```

fin

On pourrait améliorer cette création de deux manières :

- Il serait bon de vérifier si `mkdir` a réussi la création. S'il existe déjà un fichier normal qui a le nom prévu pour le répertoire de sauvegarde, ou si l'utilisateur n'a pas de droit d'écriture sur le répertoire parent, cette commande peut en effet échouer. Il faudrait alors en tenir compte pour son comportement ultérieur.
- Pour garder une certaine confidentialité aux fichiers supprimés, il serait bon de protéger le répertoire contre les accès des autres utilisateurs. On pourrait prévoir une commande `chmod 700 "$sauvegarde_rm"` après la création.

Nous pouvons commencer à présent à faire agir le script en fonction des options réclamées. Tout d'abord, les lignes 59 à 62 gèrent l'option de vidage de la corbeille :

```
59        if [ $opt_empty -ne 0 ] ; then
60            /bin/rm -rf "$sauvegarde_rm"
61            return 0
62        fi
```

Le test `[ xxx -ne yyy ]` est une comparaison arithmétique entre `xxx` et `yyy`. Il renvoie une valeur vraie si les deux éléments sont différents (*not equal*). En d'autres termes, ces lignes signifient :

si la variable représentant l'option empty n'est pas nulle, alors

effacer le répertoire de sauvegarde

quitter la fonction en indiquant une réussite

fin

La deuxième option que nous traitons est la demande d'affichage du contenu de la corbeille. Pour ce faire, nous employons les lignes 66 et 67 :

```
( cd "$sauvegarde_rm"
ls -lRa * )
```

Elles correspondent à un déplacement vers le répertoire de sauvegarde, et à un affichage récursif de son contenu et de celui des sous-répertoires. Nous avons encadré ces deux lignes par des parenthèses. On demande ainsi au shell de les exécuter dans un *sous-shell* indépendant, ce qui présente l'avantage de ne pas modifier le répertoire de travail du shell principal. Un nouveau processus est donc créé, et seul le répertoire de travail de ce processus est modifié.

L'option traitée ensuite est la récupération de fichiers ou de répertoires effacés. Une boucle s'étend des lignes 72 à 75 :

```
    while [ -n "$1" ] ; do
        mv "${sauvegarde_rm}/$1" .
        shift
    done
```

Le test `[ -n "$1" ]` vérifie si la longueur de la chaîne "`$1`" est non nulle. Ce test réussit donc tant qu'il reste au moins un argument à traiter. L'instruction `shift` que l'on trouve sur la ligne 74 sert à éliminer l'argument que l'on vient de traiter. Cette boucle permet ainsi de balayer tous les noms de fichiers ou de répertoires un par un. La ligne 73 essaie de récupérer dans le répertoire `sauvegarde_rm` un éventuel fichier sauvegardé, en le déplaçant vers le répertoire courant. Remarquons qu'il est tout à fait possible, avec cette même commande, de récupérer des arborescences complètes, y compris les sous-répertoires.

La portion du programme qui s'occupe des suppressions se situe entre les lignes 80 et 102. Elle est construite au sein d'une boucle `while-do` qui balaie les noms des fichiers et des répertoires indiqués sur la ligne de commande.

Une première étape consiste à interroger l'utilisateur si l'option `interactive` a été réclamée, et si l'option `force` ne l'a pas été. Nous voyons au passage comment deux conditions peuvent être regroupées par un *ET* logique au sein d'un test. Nous reviendrons sur ce mécanisme ultérieurement.

Après déclaration d'une variable locale, et affichage d'un message d'interrogation (avec l'option `-n` de `echo` pour éviter d'aller à la ligne), le script invoque la commande shell `read`. Cette dernière lit une ligne et la place dans la variable indiquée en argument.

Nous comparons alors cette réponse avec quatre chaînes possibles : `y`, `Y`, `o`, et `O`. L'opérateur de comparaison des chaînes est `=`, et son inverse est `!=`. Si la réponse ne correspond à aucune de ces possibilités, nous invoquons d'abord `shift`, pour décaler les arguments, puis la commande `continue` qui nous permet de revenir au début de la boucle `while`. Les arguments pour lesquels l'utilisateur n'a pas répondu par l'affirmative sont ainsi « oubliés ».

La deuxième vérification concerne les répertoires. Ceux-ci, en effet, ne peuvent être effacés qu'avec l'option `recursive`. Si le test de la ligne 92 échoue, l'argument est ignoré et on passe au suivant.

Si on a réclamé un comportement volubile du programme (option `-v` ou `--verbose`), on affiche le nom du fichier ou du répertoire traité avant de poursuivre. Puis, les lignes 100 et 101 déplacent effectivement le fichier, et passent à l'argument suivant.

La fonction qui remplace la commande `rm` est donc à présent écrite. Lorsque le script est exécuté à l'aide de l'instruction `source` ou de « . », il place cette fonction dans la mémoire du shell ; elle est prête à être invoquée à la place de `rm`. Il demeure toutefois une dernière ligne apparemment mystérieuse :

```
    trap "/bin/rm -rf $sauvegarde_rm" EXIT
```

Cette ligne est exécutée directement par le shell lorsque nous appelons notre script. La commande `trap` permet d'assurer une gestion minimale des signaux. Voici sa syntaxe générale :

```
trap commande signal
```

Lorsque le shell recevra le *signal* indiqué, il déroutera son exécution pour réaliser immédiatement la *commande* passée en argument. Les signaux permettent une communication entre processus, et surtout une prise en compte asynchrone des événements qu'un logiciel peut être amené à rencontrer (interruption d'une ligne de communication, fin d'une temporisation, dépassement d'une limite d'occupation du disque, etc.). Nous reviendrons plus en détail sur la gestion des signaux ultérieurement. Ici, la commande `trap` nous sert à intercepter un pseudo-signal simulé par le shell. En effet, avant que le shell ne se termine, il simule l'émission d'un signal nommé `EXIT`. La commande passée en argument est donc exécutée juste avant de refermer une session de travail. Elle sert à effacer le répertoire de sauvegarde.

Cette ligne est bien entendu optionnelle, mais je conseille de la conserver, car elle évite à l'utilisateur de devoir vider explicitement sa corbeille (manuellement ou d'une manière programmée à l'aide de la `crontab`).

Performances

L'utilitaire `rm_secure` n'a pas pour objet de fournir une récupération des fichiers sur du long terme. Ce rôle est dévolu aux mécanismes de sauvegarde périodique du système. En fait, `rm_secure` est conçu comme une sorte de commande *Contrôle-Z* ou *Édition/Annuler* qui permet de corriger immédiatement une erreur de frappe ayant entraîné la destruction

involontaire d'un fichier. En tenant compte de ses spécificités, chacun est toujours libre de modifier le script à sa convenance pour l'adapter à un cas particulier.

Il faut aussi être conscient que, dans un environnement graphique X Window, où plusieurs fenêtres xterm sont utilisées simultanément, dès que l'on ferme l'une d'entre elles, la commande d'effacement est invoquée, détruisant le répertoire de sauvegarde commun à toutes les fenêtres. Si ce point pose un problème, il est malgré tout possible de créer un répertoire de sauvegarde pour chaque instance indépendante du shell. On obtient cela en ajoutant au nom du répertoire un suffixe qui représente le *PID* du shell, en remplaçant la première ligne du script par :

```
sauvegarde_rm=~/.rm_saved_$$/
```

Un tel script a pour vocation de rester le plus discret possible vis-à-vis de l'utilisateur. Idéalement, on ne devrait se souvenir de sa présence que le jour où un fichier vient d'être effacé maladroitement. Il faut donc éviter de ralentir le fonctionnement de la commande `rm` originale. À cet effet, nous avons fait le choix de déplacer les fichiers avec `mv` plutôt que de les copier avec `cp`, ou des les archiver avec `tar`. Lorsque le fichier qu'il faut supprimer se trouve sur le même système de fichiers que le répertoire de sauvegarde, la commande `mv` agit instantanément, car elle ne doit modifier que le nom du fichier dans l'arborescence, et pas son contenu. Il s'agit du cas le plus courant pour les utilisateurs normaux, dont les répertoires personnels et tous les descendants se trouvent généralement sur la même partition. Pour *root*, en revanche, la situation est autre, car ce dernier doit souvent intervenir sur des répertoires qui se trouvent sur des partitions différentes. La commande `mv` ne peut plus simplement déplacer le fichier, mais est obligée de le copier, puis de supprimer l'original. Dans ce cas, notre script induit un ralentissement sensible, notamment lorsqu'on agit sur des hiérarchies complètes (par exemple, en supprimant toute l'arborescence des sources d'un programme) ou sur des fichiers volumineux (`core`, par exemple). On peut choisir de désactiver l'emploi de `rm_secure` pour *root*, en partant du principe que toute action entreprise sous ce compte est potentiellement dangereuse, et nécessite une attention accrue à tout moment.

Idéalement, la connexion *root* sur un système Unix ou Linux qui compte peu d'utilisateurs devrait être réservée à l'installation ou à la suppression de paquetages logiciels, à l'ajout d'utilisateur et à l'édition de fichiers de configuration (connexion distante, adressage réseau…). Dans tous ces cas, on n'intervient que sur des fichiers dont une copie existe « ailleurs » (CD, bande de sauvegarde, Internet). Théoriquement, *root* ne devrait jamais se déplacer – et encore moins toucher aux fichiers – dans les répertoires personnels des utilisateurs où l'on trouve des données n'existant qu'en un seul exemplaire (fichiers source, textes, e-mail…). Théoriquement…

Exemple d'exécution

Voici, en conclusion de ce survol d'un script pour shell, un exemple d'utilisation. Nous supposons que les fichiers d'initialisation (`~/.profile` ou `~/.bashrc`), exécutés lors du démarrage des sessions interactives du shell, contiennent une ligne qui permette d'invoquer le script, à la manière de `. ~/bin/rm_secure.sh`. Nous considérons donc que la

fonction `rm` définie précédemment est présente dans l'environnement, et a préséance sur le fichier exécutable `/bin/rm`.

```
$  ls
rm_secure.sh  rm_secure.sh.bak
```

Je souhaite ne supprimer que le fichier de sauvegarde, et j'appelle donc `rm *.bak` pour éviter de saisir son nom en entier. Hélas, j'introduis par erreur un caractère espace après l'astérisque.

```
$  rm * .bak
mv:.bak.: Aucun fichier ou répertoire de ce type
$  ls
$
```

Aïe ! Il ne me reste plus qu'à compter sur l'aide de notre script. Commençons par nous remémorer ses options :

```
$ rm --help
Usage: /bin/rm [OPTION]... FICHIER...
Enlever (unlink) les FICHIER(s).

  -d, --directory       enlever le répertoire, même si non vide
                        (usager root seulement)
  -f, --force           ignorer les fichiers inexistants,
                        ne pas demander de confirmation
  -i, --interactive     demander confirmation avant destruction
  -r, -R, --recursive   enlever les répertoires récursivement
  -v, --verbose         en mode bavard expliquer ce qui est fait
      --help            afficher l'aide-mémoire
      --version         afficher le nom et la version du logiciel
Rapporter toutes anomalies à <bug-fileutils@gnu.org>.
rm_secure:
  -e --empty      vider la corbeille
  -l --list       voir les fichiers sauvés
  -s, --restore   récupérer des fichiers
$  rm -l
-rwxr-xr-x cpb/cpb     2266 2007-09-01 19:39:14 rm_secure.sh
-rwxr-xr-x cpb/cpb     2266 2007-09-01 19:39:14 rm_secure.sh.bak
$ rm --restore rm_secure.sh
$ ls
rm_secure.sh
```

Ouf ! Le fichier est récupéré. Vérifions maintenant que le répertoire de sauvegarde est bien vidé automatiquement lors de la déconnexion :

```
$ rm --list
-rwxr-xr-x cpb/cpb    2266 2007-09-01 19:39:14 rm_secure.sh.bak
$ exit
```

Essayons de voir s'il reste des fichiers après une nouvelle connexion :

```
$ rm --list
ls: *: Aucun fichier ou répertoire de ce type
$
```

Ce dernier message n'est peut-être pas très heureux. Nous laisserons au lecteur le soin d'encadrer l'appel `ls` de la ligne 67 par un test `if-then-fi` qui affiche un message plus approprié si la commande `ls` signale une erreur (ne pas oublier de rediriger la sortie d'erreur standard de cette dernière commande vers `/dev/null` pour la dissimuler).

Conclusion

Avec ce chapitre d'introduction à la programmation shell, nous avons pu observer la structure des scripts, que nous pourrons étudier plus en détail par la suite.

On peut remarquer la concision de ce langage, qui permet de réaliser une tâche déjà respectable en une centaine de lignes de programmation. Et ce, en raison du niveau d'abstraction élevé des commandes employées. Par exemple, la suppression récursive d'un répertoire, comme nous l'avons vu dans la dernière ligne du script, demanderait au moins une vingtaine de lignes de programmation en C.

Les chapitres à venir vont nous permettre d'étudier la programmation shell d'une manière plus formelle, et plus complète.

Exercices

2.1 – Appel d'un script par « source » (facile)

Créez un script qui initialise une nouvelle variable avec une valeur arbitraire, puis affiche cette variable.

Lancez le script (après l'avoir rendu exécutable avec `chmod`) comme nous l'avons fait dans le chapitre 1. Une fois l'exécution terminée, essayez d'afficher le contenu de la variable :

initialise_et_affiche.sh

```
#! /bin/sh
```

```
ma_variable=2007
echo $ma_variable
```

```
$ chmod 755 initialise_et_affiche.sh
$ ./initialise_et_affiche.sh
2007
$ echo $ma_variable
```

Résultat ?

Essayez à présent d'exécuter le script avec l'invocation suivante :

```
$  .  initialise_et_affiche.sh
```

Et essayez ensuite de consulter le contenu de la variable.

N. B. : vous pouvez remplacer le « `.` » de l'invocation par le mot-clé « `source` » si vous travaillez avec `bash`.

2.2 – Utilisation de rm_secure.sh (plutôt facile)

Éditez le fichier de personnalisation de votre shell de connexion (`.bashrc`, `.profile`...) et insérez-y la ligne :

```
.  ~/bin/rm_secure.sh
```

Prenez soin de placer le script `rm_secure.sh` dans le sous-répertoire `bin/` de votre répertoire personnel.

Reconnectez-vous, et essayez à présent d'appeler :

```
$ rm -v
```

pour vérifier si le script est bien appelé à la place de la commande `rm` habituelle.

Effectuez quelques essais d'effacement et de récupération de fichiers.

2.3 – Adaptation de rm_secure.sh (plutôt difficile)

En examinant le contenu du script `rm_secure.sh`, essayez d'y apporter quelques modifications (par exemple modifiez le répertoire de sauvegarde) et améliorations (par exemple la confidentialité sera augmentée si le script retire aux fichiers sauvegardés tous les droits d'accès, hormis à leur propriétaire).
